Appendix L

Simulating Arbitrary-Precision Fixed-Point and Floating-Point Roundoff in Matlab

Matlab is a natural environment for scientific and engineering simulations. Its workspace concept, interactivity, easy programming and visualization capabilities all significantly facilitate the user's task.

There are different levels to simulate roundoff. Here we will briefly discuss four such possibilities:

- simple, straightforward programming of approximate roundoff,
- use of advanced quantizers and roundoff tools freely available from the site of this book,[1]
- use of the quantized DSP simulation toolbox, freely available from the site of this book,[1]
- use of the Fixed-Point Toolbox, commercially available from The MathWorks.[2]

The basic idea of the first two possibilities is to evaluate each operation in Matlab, and follow each by a roundoff operation. For example, if a and b are two finite wordlength numbers which are to be multiplied, the result is evaluated as $c = Q(a * b)$. This is a fast and simple approach, and generally provides good results. There are, however, some limitations which will be discussed below.

The last two possibilities are based on objects. The methods provided by the object libraries allow precise simulation of virtually any finite wordlength operation occurring in practice, at the cost of slower execution.

[1] `http://www.mit.bme.hu/books/quantization/`

[2] `http://www.mathworks.com/`

L.1 STRAIGHTFORWARD PROGRAMMING

L.1.1 Fixed-point roundoff

Fixed-point roundoff is simplest to simulate in Matlab by executing

```
xq=q*round(x/q);
```
(L.1)

where q is the quantum size. Matlab which uses IEEE double-precision representation of numbers, executes calculations "infinitely precisely", compared to step size q. In many cases, this is sufficient: operation (L.1) returns the multiple of q closest to the result.

This method is simple and straightforward. However, several possible properties of roundoff cannot be treated by it, like

- amplitude limits (saturation or overflow),
- bit numbers over 53,
- treating values at equal distance from two representable values in a selected way,
- different number representations,
- quantization replaced for simulation purposes by PQN,
- different roundoff strategies,
- etc.

To treat these properties, more advanced methods are necessary, as to be discussed in Section L.2 and further.

L.1.2 Floating-Point Roundoff

In floating-point rounding, it is not the quantum size which is fixed, but rather the precision of the mantissa (see page 343), that is, its number of bits. A simple Matlab implementation of floating-point quantization with any p less than 53 is:[3]

$$\begin{aligned}&\texttt{[f,e]} = \texttt{log2(x);}\ \ \texttt{dxp} = \texttt{sign(x)}.*\texttt{pow2(max(e,-1021)} + \texttt{52} - \texttt{p);}\\&\texttt{xqfl} = \texttt{(x} + \texttt{dxp)} - \texttt{dxp;}\end{aligned}\tag{L.2}$$

[3]This is implemented in the function M-file `roundfloat.m`, available from the book's web page. `roundfloat.m` works not only for real numbers, but also for complex numbers and arrays.

For $p = 53$ one uses Matlab itself. For $p > 53$, Matlab simulation of floating-point quantization is possible but it is rather complicated to implement (see Section L.3).

Algorithm (L.2) is simple and straightforward. However, there are some limitations, like:

- albeit rarely, errors can be slightly larger than half of the quantum size, the error limit required by the IEEE standards. This phenomenon is called "double rounding," see Example L.1 in page 716.
- it works only for precisions $p \leq 53$,
- quantization cannot be replaced for simulation purposes by PQN,
- the roundoff strategy cannot be easily modified,
- the range of the possible values of the exponent is limited by the range of the exponents of the IEEE double-precision number representation,
- etc.

If these properties are important, more advanced tools have to be used.

L.2 THE USE OF MORE ADVANCED QUANTIZERS

In Matlab, numbers are handled and stored in IEEE double-precision representation. When simulation is executed making use of these numbers, numbers rounded by the arithmetic processor are re-quantized to conform with the desired bit length.

However, there is a small problem in re-quantization to fixed-point, when the probability of the occurrence of the input values $(k + 0.5)q$ is not negligible[4] (this is the case e.g. when executing the FFT with block-float number representation, see e.g. page 394). Matlab's `round` operation rounds every number at $(k + 0.5)$ upward for positive numbers, and downward for negative numbers. Quantization theory, on the other hand, tacitly assumes that for these numbers with a tie of the distance to the two neighboring quantized values, quantization happens with probability 0.5 in each direction.

Rounding which corresponds to the latter (in a pseudo-randomized way) can be simulated for fixed-point by the Matlab function `roundrand`, written for this book, and available from the book's web page. Randomization, however, has the disadvantage that it is irreproducible unless the random generator is reset before each run of the same procedure.

Another approach is as follows. In some DSP processors, the clever algorithm of *convergent rounding* (see page 396) is executed: numbers with two possible roundoff results in equal distance are rounded to the nearest number *with LSB zero*.

[4] k is an integer.

x	representation	x' representation	x'
-15	$-1.111 \cdot 2^3$	$-1.00 \cdot 2^4$	-16
-2	$-1.000 \cdot 2^1$	$-1.00 \cdot 2^1$	-2
-1	$-1.000 \cdot 2^0$	$-1.00 \cdot 2^0$	-1
0	$+0.000 \cdot 2^{\mathrm{Emin}}$	$+0.00 \cdot 2^{\mathrm{Emin}}$	0
0.5	$+1.000 \cdot 2^{-1}$	$+1.00 \cdot 2^{-1}$	0.5
0.8125	$+1.101 \cdot 2^{-1}$	$+1.10 \cdot 2^{-1}$	0.75
0.875	$+1.110 \cdot 2^{-1}$	$+1.11 \cdot 2^{-1}$	0.875
0.9375	$+1.111 \cdot 2^{-1}$	$+1.00 \cdot 2^0$	1
1	$+1.000 \cdot 2^0$	$+1.00 \cdot 2^0$	1
7	$+1.110 \cdot 2^2$	$+1.11 \cdot 2^2$	7
8	$+1.000 \cdot 2^3$	$+1.00 \cdot 2^3$	8
9	$+1.001 \cdot 2^3$	$+1.00 \cdot 2^3$	8
10	$+1.010 \cdot 2^3$	$+1.01 \cdot 2^3$	10
11	$+1.011 \cdot 2^3$	$+1.10 \cdot 2^3$	12
12	$+1.100 \cdot 2^3$	$+1.10 \cdot 2^3$	12
13	$+1.101 \cdot 2^3$	$+1.10 \cdot 2^3$	12
14	$+1.110 \cdot 2^3$	$+1.11 \cdot 2^3$	14
15	$+1.111 \cdot 2^3$	$+1.00 \cdot 2^4$	16

TABLE L.1 Convergent rounding of a few numbers to representation with $p = 3$.

This can be simulated for fixed-point by the Matlab function `roundconv`, written for this book and also available from the book's web page. The formula (L.2) executes convergent rounding for floating-point.

Convergent rounding can be understood by examining Table L.1, evaluated for a few numbers with (L.2), with $p = 3$.

For the fine-tuning of quantizers, their properties can be defined and used in the functions of the `roundoff` toolbox.[5] Here are the basic properties:

```
%Quantizer properties and values:
  type = type of quantizer ('fixed-point' or 'floating-point')
  rdir or roundingdirection = direction of rounding:
         'r', 'f', 'c' 'x', 'i', 't'
         for round, floor, ceil, fix, toinf, trunc
```

[5]See `http://www.mit.bme.hu/books/quantization/`

```
  treath or treathalf = way of treating *.5 values for round:
         'f', 'c', 'x', 'e', 'o', 'r'
         for floor, ceil, fix, toinf, toeven, toodd, rand
  coding = coding of mantissa, 's', 'o', 't'
         for sign-magnitude, one's complement, two's complement
  dither = adding dither, 'on' or 'off'
  operation = way of operation, 'quantize' or 'PQN' to add PQN
Fixed-point quantizers only:
  B or bits = number of bits of the number, sign included
  xnmin or minimum = minimum representable (negative) number
  xpmax or maximum = maximum representable number
  fb or fractionbits = number of fractional bits
  ovf or overflow = overflow handling, 'c', 'm', 't', 'i', 'n'
        for clip, modular, triang, inf, none
Floating-point quantizers only:
  p or precision = precision number of bits of the significand,
      sign not included, but hidden bit is included
  lb or leadingbit = leading bit representation, 'h' or
      'hidden' for hidden, 's' or 'shown' for shown
  ufl or underflow = underflow, 'g' for gradual, 'f' for flush
  Emin or Expminimum = min. value of exponent (al least -1022)
  Emax or Expmaximum = max. value of exponent (at most 1023)
  Ebias or Expbias = bias in exponent representation
```

An example of the use is as follows:

```
  Q = qstruct('p',24); %define a quantizer
  a = roundq(1.1,Q); b = roundq(2.3,Q); %rounded input numbers
  c = roundq(a*b,Q); %operation and subsequent roundoff
  num2bin(c,Q) %show contents of binary number
  %
  Q_PQN=qstructset(Q,'operation','PQN'); %replace Q by PQN
  c2 = roundq(a*b,Q_PQN); %operation and addition of PQN
  num2bin(c2,Q_PQN) %show contents of binary number
```

This tool is quite versatile, and can be used to simulate many practical cases. However, for accurate simulation of DSP hardware, there are some difficulties which can be overcome with object-based programming only, at the price of slower execution of the more complicated code. We enumerate three of these here.

The first problem is that even with IEEE double precision, the result of not all operations can be precisely obtained. The small roundoff error in double-precision execution can be enough to bring the result to the border of two quantum bins in the number representation, and subsequent roundoff of a number on the border may act in the same direction as with double precision, increasing the total error slightly over half of the quantum size.

Example L.1 Erroneous Result Due to Double Rounding
Let us evaluate in floating-point, precision $p = 28$, the following expression:

$$1.0000\,0000\,0000\,0000\,0000\,0000\,001{\cdot}2^3{+}1.1111\,1111\,1111\,1111\,1111\,1111\,000{\cdot}2^{-26}$$

The mathematically precise result is

$$1.0000\,0000\,0000\,0000\,0000\,0000\,001\,|\,0\,1111\,1111\,1111\,1111\,1111\,1111\,1{\cdot}2^3.$$

In IEEE double precision (Matlab's native number representation), this is represented with precision $p = 53$, that is, it needs to be rounded to the nearest representable number:

$$1.0000\,0000\,0000\,0000\,0000\,0000\,001\,|\,1\,0000\,0000\,0000\,0000\,0000\,0000{\cdot}2^3.$$

The quantizer with precision $p = 28$ rounds this, according to the convergent rounding rule (see page 396), to

$$1.0000\,0000\,0000\,0000\,0000\,0000\,010{\cdot}2^3.$$

The proper result would be the representable number closest to the precise result:

$$1.0000\,0000\,0000\,0000\,0000\,0000\,001{\cdot}2^3$$

The second problem is that precision of this solution is limited to $p = 53$, thus accumulator wordlengths larger than this cannot be simulated, moreover, the roundoff of IEEE double precision cannot be evaluated due to lack of higher-precision reference values.

The third problem is that the implementation of each roundoff needs the explicit call of the rounding routine. This is tedious to implement, and is prone to certain roundings left out from simulation.

L.3 QUANTIZED DSP SIMULATION TOOLBOX (QDSP)

The most precise freely available tool is the so-called Quantized DSP Simulation toolbox.[6] It makes use of the possibility of using objects in Matlab.

First, a few quantizers are defined.

```
qmake('name','Qs','type','floating-point','precision',24);
qmake('name','Qd','type','floating-point','precision',53);
qmake('name','Q_PQN','precision',24,'operation','PQN');
```

[6]See `http://www.mit.bme.hu/books/quantization/`

Then the algorithm is described in the usual way as algorithms are described in Matlab, like

```
u=0.25*sin([1:100]/100*2*pi*5);
Q='Qs';
%Define coefficients and inputs as qdata objects:
b1=qdata(0.6,Q); a1=qdata(0.8,Q); a2=qdata(0.21,Q);
uq=qdata(u,Q); y=qdata(zeros(size(u)),Q);
%
for k=3:length(uq)
   y(k)=-a1*y(k-1)-a2*y(k-2)+b1*uq(k-1);
end
ysingle=y;
```

Now by changing `Q` to `Q='Qd'`, and writing the result into `ydouble`, the difference of the `y`'s (`ydouble-ysingle`) gives the difference between evaluation of the algorithm in single precision and double precision, without changing anything in the program of the algorithm. `Q='Q_PQN'` allows running the algorithm with PQN added in the place of each quantization, allowing the verification of quantization theory.

In order to simulate DSP's with accumulator bitlength exceeding memory bitlength, the accumulator may be predefined with its proper bitlength. At each operation, the bit length of the left operand is retained. Meanwhile, assignment to an element of a vector retains the properties of the quantizer of the target: this allows direct storage "to the memory":

```
qmake('name','Qm','type','fixed-point','Bits',16,'fract',15);
qmake('name','Qacc','type','fixed-point','Bits',40,'fract',30);
u=0.25*sin([1:100]/100*2*pi*5);
%Define coefficients and inputs as qdata objects:
b1=qdata(0.6,'Qm'); a1=qdata(0.8,'Qm'); a2=qdata(0.21,'Qm');
uq=qdata(u,'Qm'); y=qdata(zeros(size(u)),'Qm');
%Increase precision of coefficients to the accumulator:
b1=qdata(b1,'Qacc'); a1=qdata(a1,'Qacc'); a2=qdata(a2,'Qacc');
%
%This is the simulation cycle:
for k=3:length(uq)
  y(k)=-a1*y(k-1)-a2*y(k-2)+b1*uq(k-1);
end
```

More details are available by running the toolbox (e.g. and typing `'help qdsp'`, or `'qdspstart'`), or from the home page

`http://www.mit.bme.hu/books/quantization/Matlab-files.html.`

L.4 FIXED-POINT TOOLBOX

For those who have access to the Fixed-Point Toolbox which is commercially available with Matlab, similar possibilities are available for fixed-point number representation as described in Section L.3. In Matlab, see `'help fixedpoint'`, or `'doc fixedpoint'`, or see the general description of the toolbox.

The example shown in Section L.3 can be executed with 16-bit fixed-point numbers in the interval $(-1, 1-2^{-15})$ (16 bits, 15 fractional bits, two's complement number representation) as given below.

```
Q16=quantizer(struct('mode','fixed','format',[16,15],...
                    'roundmode','convergent'));
u=0.25*sin([1:100]/100*2*pi*5);
b1=fi(0.6,Q16); a1=fi(-0.8,Q16); a2=fi(-0.21,Q16);
uq=fi(u,Q16); y=fi(zeros(size(u)),Q16);
%
for k=3:length(uq)
  %round results to the prec. of y by subscripted assigment:
  y(k)=a1*y(k-1);
  y(k)=y(k)+a2*y(k-2);
  y(k)=y(k)+b1*uq(k-1);
end
yfix=y;
```

The result can be directly compared to the result of the double-precision calculation:

```
u=0.25*sin([1:100]/100*2*pi*5);
b1=0.6; a1=0.8; a2=0.21;
y=zeros(size(u)); uq=u;
%
for k=3:length(uq)
  y(k)=-a1*y(k-1);
  y(k)=y(k)-a2*y(k-2);
  y(k)=y(k)+b1*uq(k-1);
end
ydouble=y;
erry=double(yfix)-ydouble;
```

Notice that the algorithm is executed with exactly the same code in both cases. This offers a reliable testing method for algorithms.

It is also possible to use the increased precision of the accumulator:

```
%Type definitions:
Qmem=quantizer('fixed',[16,15],'convergent');
racc=fimath; racc.OverflowMode='wrap';
```

```
racc.ProductMode='KeepLSB';
racc.ProductWordLength=40;
racc.SumMode='KeepLSB';
racc.SumWordLength=40;

%Variable definitions:
u=0.25*sin([1:100]/100*2*pi*5);
b1=fi(0.6,Qmem,'fimath',racc);
a1=fi(-0.8,Qmem,'fimath',racc);
a2=fi(-0.21,Qmem,'fimath',racc);
uq=fi(u,Qmem,'fimath',racc);
y=fi(zeros(size(uq)),Qmem,'fimath',racc);

%Algorithm
for k=3:length(uq)
  y(k)=a1*y(k-1)+a2*y(k-2)+b1*uq(k-1);
end
yfix=y;
```

More examples are available from the home page

`http://www.mit.bme.hu/books/quantization/Matlab-files.html.`